IAC-15,B4,6B,12,x30427

HIGH INTEGRITY SOFTWARE FOR CUBESATS AND OTHER SPACE MISSIONS

Dr. Carl Brandon, Professor and Director
CubeSat Lab, Vermont Technical College, Randolph Center, VT, USA carl.brandon@vtc.edu

Dr. Peter Chapin, Professor and Software Director
CubeSat Lab, Vermont Technical College, Randolph Center, VT, USA peter.chapin@vtc.edu

ABSTRACT

We currently have an operating single CubeSat launched as part of NASA's ELaNa IV program on November 19, 2013, the first satellite of any kind launched by a college in New England. Many CubeSat failures have been attributed to software failures. Of the twelve university CubeSats that were launched with ours, we are the only one that is functional. Two had partial contact for a week, one lasted four months, and eight were never heard from. These other CubeSats primarily used the C language. We are using the most reliable software technology ever sent into space. We used the SPARK 2005 Toolset and Ada language in the construction of our software. Ada is used in almost all European Space Agency and many NASA rockets and spacecraft, and in most European rail systems and nuclear power plants. SPARK is used in commercial aviation (Rolls-Royce Trent jet engines, ARINC ACAMS system, Lockheed Martin C130J), military aviation (EuroFighter Typhoon, Harrier GR9, AerMacchi M346), air-traffic management (UK NATS iFACTS system), rail (numerous signaling applications), and medical (LifeFlow ventricular assist device) applications.

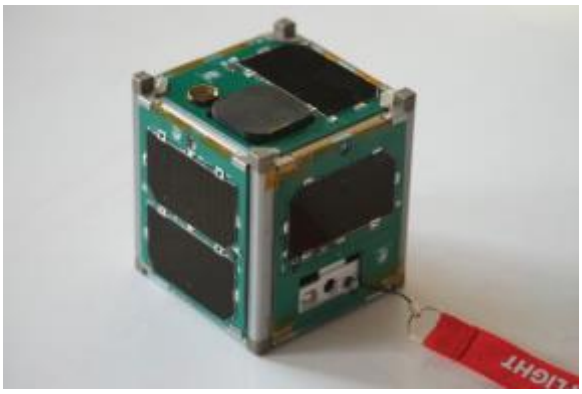
We are using SPARK/Ada, with its reduction of errors by a factor of about 100 compared with C. SPARK is a formally defined programming language and a set of verification tools specifically designed to support the development of high integrity software, and can formally verify information flow, freedom from runtime errors, functional correctness, and security and safety policies.

Ours is the first spacecraft to use SPARK. We are currently upgrading our CubeSat software to SPARK 2014, and will then work on improving some of the algorithms in that software. We would then have a very reliable software platform, CubedOS that other projects could use as a base for their CubeSat or other spacecraft projects. Our next CubeSat, Lunar IceCube with Morehead State University (PI), Goddard Space Flight Center (BIRCHES & Lunar transfer trajectory) and the Jet Propulsion Lab (Iris 2) is self propelled with a Busek iodine ion drive which will go to the Moon on the Space Launch System EM-1 flight in 2018. This software will be much more complex, dealing with power management, the ADACS, infrared spectrometer (BIRCHES), the data and navigation radio (Iris 2), the electrical power system, and aim the photo voltaic panels and ion thruster. The software will carry out the navigation plan and deal with ground based commands and upgrades. SPARK's reliability will be necessary for this.

VERMONT LUNAR CUBESAT DESCRIPTION

Hardware

The Vermont Lunar CubeSat is a 1U (10cm x 10cm x 10cm with 0.7cm and 0.65cm legs on the ends, maximum mass of 1.33kg) CubeSat. Our mass is 1.01kg. It uses the aluminum structure and CPU board, using the Texas Instrument MSP430 processor, from CubeSat Kit. We used a commercial electrical power system (EPS) from Clyde Space in Glasgow, Scotland. This supplied regulated 3.3V and 5V and raw 8.4V from the 10Wh battery. It also had three charge controllers for the six photovoltaic panels on the sides of the Cubesat. A deployable crossed yagi antenna system from ISIS in Delft, Netherlands, for the 70cm and 2m bands, used by our radio, a Helium-100 from Astrodev with a 2m receiver for uplink and a 70cm transmitter for downlink. Our GPS receiver from Novatel was mounted on an interface board from Astrodev (designed at the University of Michigan). We made a board with connectors for the LEDs and mounting for a GPS antenna with a 33dB LNA and a VGA camera with built in JPEG compression. The outer surface of the satellite had six boards with 29% efficient photovoltaic cells, four panels with two each and two panels with one each. All of the panels also had two high power green LEDs each.



Vermont Lunar CubeSat before launch.

Software

Our software for the Vermont Lunar CubeSat is written primarily in SPARK[1] 2005 (discussed below). Some of the software metrics are:
5991 lines of code.
4095 lines of comments (2843 are SPARK annotations).
A total of 10,086 lines (not including blank lines).
The Examiner generated 4542 verification conditions, all but 102 were proved automatically (98%).
We attempted to prove the program free of runtime errors, which allowed us to suppress all checks.
The C portion consisted of 2239 lines (including blank lines).
Additional provers in SPARK 2014[1] would allow 100% automatic proofs.

Mission

Our CubeSat was designed primarily as a technology demonstrator for navigation components for a hoped for future Lunar mission. As described below, we are now working on that mission. Other important aspects were to gain experience in all aspects of satellite construction, launch and operation. In addition, we wanted to show

the superior reliability of the SPARK/Ada software technology. We applied for and were accepted in the first group of the NASA Educational Launch of Nano-Satellites (ELaNa), and became part of the ELaNa IV launch. This was a flight arranged by NASA on an Air Force Minotaur 1, the ORS-3 launch, to a 500km altitude, 40.5 degree inclination circular orbit. The launch occurred on November 19, 2013, from Wallops Island, Virginia. On this launch were 14 Air Force CubeSats, 2 NASA CubeSats and 12 university CubeSats in addition to an Air Force TAC-3 larger satellite.

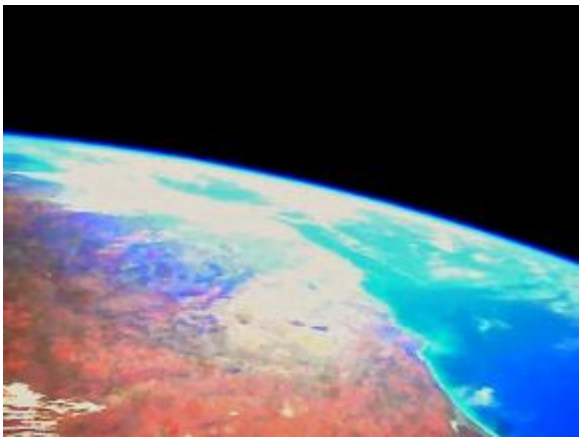
Our ORS-3 launch from Wallops Island.

ELaNa IV results

The Air Force satellites appear to be mostly successful, but due to the classified nature of some, we don't have the details. Both of the NASA CubeSats were successful.

The university CubeSats were another story. Eight of the twelve were never heard from. One fried their batteries on the first day due to a couple of software errors. One had partial contact for less than a week. One worked successfully for four months. Ours is still fully operational at 22 months, having travelled 10,000 orbits and 266 million miles (428 million kilometers). All of the other university CubeSats had software primarily written in C.

In addition to inertial measurement data, we can also command our CubeSat to take photos and GPS data. Our first photo of Australia in 2014, and a recent photo from June 2015 are below.



North coast of Western Australia

Clouds over the ocean

PAST ADA AND SPARK USAGE

Arctic Sea Ice Buoy

Our first project using SPARK/Ada was a sub-contract to build a prototype Arctic Sea Ice Buoy with GPS, temperature and wind speed and direction sensors. It used the same CubeSat Kit CPU board that was later used in the Vermont Lunar CubeSat. This was much simpler software then was later used in our CubeSat. It allowed us to gain experience with SPARK/Ada, and develop the software tool chain that we later used in our CubeSat. There was no Ada compiler for the MSP430, so we wrote the software in SPARK/Ada, then ran it through AdaMagic (the front end of an Ada compiler which uses ANSI standard C as the intermediate language). The C was then compiled with a C compiler. This resulted in the C software being proved correct, as it was translated from the SPARK/Ada source code.

SPARK DESCRIPTIONS AND CHARACTERISTICS

Toolset

The current version of the SPARK toolset and language definition is SPARK 2014. It is a major enhancement over the earlier SPARK 2005 toolset and language definition we briefly described in Section Vermont Lunar CubeSat – Software above. The SPARK 2014 language supports a much larger subset of Ada, allowing more natural designs. The SPARK 2014 toolset uses more modern theorem provers, and is more easily extensible to use additional provers as they become available. The net effect of these enhancements is that SPARK 2014 is much easier to use, allowing the developer to focus more on the problem being solved and less on working around the idiosyncrasies of the programming environment.

In this section we give an overview of the SPARK 2014 toolset and language so the reader can better understand the nature of SPARK programming and the advantages it offers. For a more complete description of SPARK 2014 see, for example, McCormick-Chapin-2015[1].

All current and future software development done by the CubeSat Laboratory at Vermont Technical College, including the work described in Section New Work below, is being done using SPARK 2014. Unless otherwise stated all following uses of SPARK in this paper refer to SPARK 2014.

SPARK 2014 - Toolset

The SPARK tools consist of a modified Ada compiler together with a verification condition generator and one or more back-end theorem provers.

Adacore's GNAT Ada compiler has been modified to understand the additional SPARK aspects, described in the next section, and to verify, upon request, conformance to the restrictions of the SPARK language. Certain diagnostic messages produced by "the SPARK tools" are actually produced by the modified Ada compiler before the specialized tools are run. These are typically messages related to the structure of the program (i.e., syntax errors in the SPARK specific constructs).

An additional tool, GNATprove, performs detailed data and information flow analysis, described in the next section, and generates verification conditions for the provers. Conceptually GNATprove produces a verification condition, or "check," for every place where the Ada language mandates a runtime check. If these verification conditions are proved, or "discharged," it

means the runtime check will never fail. Examples of such runtime checks include: out of bounds array access, arithmetic overflow, division by zero, and some other things.

In addition the Ada language allows the programmer to express range constraints on values to ensure the results of computations are always in an appropriate range (e.g., never negative, always in the range 1 to 100, etc.) Ada normally includes runtime checks to verify these constraints; GNATprove generates verification conditions that, if discharged, will statically show they never fail.

Furthermore Ada 2012 allows the programmer to include pre- and postconditions on subprograms, as well as other assertions, that encode higher level correctness properties (e.g., a sort procedure produces a sorted permutation of its input). Again, GNATprove generates verification conditions that, if discharged, will statically show those properties will always hold.

At the time of this writing the SPARK tools ship with two back-end theorem provers, Alt-Ergo[2] and CVC4[3]. Two provers are used to take advantage of their complementary strengths; verification conditions unprovable by one prover might be handled by the other. It is possible to configure the SPARK tools to use only one prover or additional provers obtained separately, such as Microsoft's Z3[4].

The GPS integrated development environment developed by Adacore provides a convenient front-end to the SPARK tools. Using the tools can be as easy as selecting "Prove File" from the GPS menus. The result is a list of locations where unproved verification conditions exist, if any. The programmer can then view and edit those locations as necessary.

Proofs fail for three reasons:

- The code is incorrect. The check being analyzed might actually fail.
- The theorem prover(s) are not powerful enough to complete the proofs.
- There is insufficient information in the program to complete the proofs.

Most of the skill in using the SPARK tools is in determining which of these cases is the problem, and in modifying the program to deal with that situation.

It is important to understand that the GNAT Ada compiler can insert runtime checks for all the SPARK assertions as well as the Ada language mandated checks. During testing it would be typical to build the program with these runtime checks enabled. Thus checks that can't be completely proved can still be tested. Once all checks are proved, the runtime checking can be disabled, saving both space and time in the final program without compromising safety.

Language

The SPARK language is a subset of Ada in that certain Ada features that are difficult to analyze using current technology have been removed from the language. Specifically SPARK supports neither exception handling nor access types (pointers). In SPARK it is necessary to report errors using returned status values. However, \SPARK's flow analysis ensures that all
such values are checked. It is not possible to ignore error codes in a SPARK program that passes examination without warning.

The lack of access types may seem more limiting but Ada, in general, requires less use of explicit indirection than is typical in C programs. In Ada, and in \SPARK, arrays are first class citizens of the language and can be passed into and returned from subprograms directly.

Also arrays can be dynamically sized on the stack without the use of an explicit memory allocator.

The SPARK language also extends Ada with additional aspects that enrich declarations and

additional assertions that describe conditions that must hold true in every execution of the

program. The additional aspects include data dependency and information dependency declarations.

The additional assertions include pre- and postconditions, loop invariants, subtype predicates, and other related things.

As an example consider the following specification of a SPARK package containing a single global datebook object along with subprograms for manipulating it:

```
with Dates;

use type Dates.Datetime;

package Datebook

  with

    SPARK_Mode => On,

    Abstract_State => State

is

   Maximum_Number_Of_Events :
constant := 64;

   subtype Event_Count_Type is
Natural range 0 ..
Maximum_Number_Of_Events;



   type Status_Type is (Success,
Description_Too_Long,
Insufficient_Space, No_Event);



   -- Initializes the datebook.

   procedure Initialize

   with

     Global => (Output => State),

     Depends => (State => null);

   -- Adds an event to the
datebook.

   procedure Add_Event

     (Description : in  String;

      Date : in  Dates.Datetime;

      Status : out Status_Type)

   with

     Global => (In_Out => State),

     Depends => (State =>+
(Description, Date), Status =>
(Description, State));

   -- Other subprograms as
required...

end Datebook;
```

The package is decorated with a `SPARK_Mode` aspect set to `On` indicating that this compilation unit is intended to abide by the restrictions of the SPARK language. The fact that the package contains internal global state is declared explicitly using the `Abstract_State` aspect. How that internal state is manipulated by the subprograms is also declared explicitly using the `Global`

and `Depends` aspects. For example, the `Add_Event` procedure both reads and writes the global state. Specifically the new state depends on itself (the meaning of the plus sign in the =>+ notation) and on the `Description` and `Data` parameters.

The SPARK tools use this information to verify that all values are initialized before use and that all computed results are used in some way. For example, calling `Add_Event` before calling `Initialize` is detected because `Add_Event` reads the package state and thus requires it to be initialized first. Similarly since `Status` is an out parameter of the procedure the SPARK tools will verify that its value is used in some way; ignoring status codes is not allowed.

The SPARK tools will further verify that the dependency declarations are supported by the implementation in the package body (not shown here for the sake of brevity).

As another example consider the following specification of a SPARK package containing a search procedure:

```
package Searchers

  with SPARK_Mode => On

is

   subtype Index_Type is Positive
range 1 .. 100;

   type Array_Type is
array(Index_Type) of Integer;

   procedure Binary_Search
(Search_Item : in  Integer;

     Items : in  Array_Type;

     Found : out Boolean;

                              Result
: out Index_Type)

      with

         Pre =>

            (for all J in
Items'Range =>

               (for all K in J + 1
.. Items'Last => Items(J) <=
Items(K))),

         Post =>

           (if Found then
Search_Item = Items(Result)

                     else (for all
J in Items'Range => Search_Item /=
Items(J)));

end Searchers3;
```

Following normal Ada style, an array type is defined that is indexed over a subrange of the range of positive integers. The `Binary_Search` procedure takes an item to search for, an array to search, and outputs a Boolean flag to indicate if the item is found along with the item's location in the array if it is.

The procedure declaration is enhanced with additional semantic information in the form of pre-and postconditions. The precondition states that the input array is sorted. The postcondition states that if the item is found the returned index is, in fact, the location of the item. On the other hand if the item is not found, it does not exist in the array.

The body of this package showing the implementation of the procedure is:

```
package body Searchers

  with SPARK_Mode => On

is

   procedure Binary_Search
(Search_Item : in  Integer;
```

```
   Items  : in  Array_Type;

   Found  : out Boolean;

   Result : out Index_Type) is

      Low_Index : Index_Type :=
      Items'First;

      Mid_Index  : Index_Type;

      High_Index : Index_Type :=
      Items'Last;

   begin

      Found  := False;

      Result := Items'First;
-- Initialize Result to "not found"
case.

      -- If the item is out of
range, it is not found.

      if Search_Item <
      Items(Low_Index) or
      Items(High_Index) <
      Search_Item then

         return;

      end if;

      loop

         Mid_Index := (Low_Index +
High_Index) / 2;

         if Search_Item =
Items(Mid_Index) then

            Found  := True;

            Result := Mid_Index;

            return;

         end if;



         exit when Low_Index =
High_Index;


         pragma Loop_Invariant (not
Found);

         pragma Loop_Invariant
(Mid_Index in Low_Index ..
High_Index - 1);

         pragma Loop_Invariant
(Items(Low_Index) <= Search_Item);

         pragma Loop_Invariant
(Search_Item <= Items(High_Index));

         pragma Loop_Variant
(Decreases => High_Index -
Low_Index);


         if Items(Mid_Index) <
Search_Item then

            if Search_Item <
Items(Mid_Index + 1) then

               return;

            end if;

            Low_Index := Mid_Index
+ 1;

         else

            High_Index :=
Mid_Index;

         end if;


      end loop;

   end Binary_Search;


end Searchers;
```

The SPARK tools will first generate verification conditions at each place in the body where an Ada check is required. For example every place where the |Items| array is accessed must be checked to ensure the index used is in range. Using the precondition as an initial hypotheses, and adding information based on the actions taken in the procedure, the SPARK tools

will generate a verification condition to show that the postcondition is always true. Furthermore at every call site a verification condition will be generated to show that the precondition must be true at that call site.

In this example all of these verification conditions are proved automatically showing that the procedure is free of unexpected runtime errors **and** that it always honors its strong postcondition (given the precondition).

The `Loop_Invariant` pragmas in the procedure where written to assist the proving process. They

represent conditions that must be true at that point for every iteration of the enclosing loop.

The SPARK tools prove that the invariants are true on the first iteration and that they remain true on all following iterations. The tools can then use the conditions in the invariants to complete following proofs, such as the postcondition in this case.

The `Loop_Variant` pragma is used to prove that the loop will eventually terminate. It gives an expression that, in this case, always decreases with each loop iteration. Because the types involved are bounded and because the SPARK tools have already proved that overflow errors are impossible, even in the assertion expressions themselves, it follows that the loop must end since the value of a bounded expression can't decrease forever.

Although this example can only search arrays of 100 integers, it is possible, although admittedly more difficult, to write general purpose code that is similarly proved free of errors. Overall these examples only give a flavor of SPARK and many features and details have been left out for the sake of brevity.

WHERE SPARK WOULD HAVE HELPED

Ariane 5 initial flight failure

The Ariane 5 software was reused from the Ariane 4, written in Ada. The greater horizontal acceleration in the larger Ariane 5 caused a data conversion from a 64-bit floating point number to a 16-bit signed integer value to overflow and cause a hardware exception. "Efficiency" considerations had omitted range checks for this particular variable, although conversions of other variables in the code were protected. The exception halted the gyro reference platforms, resulting in the destruction of the flight. The financial loss was over $500,000,000. SPARK/Ada would have prevented this failure

Boeing 787 generator control computer

There are two generators for each of the two engines, each with its own control computer programmed in Ada. The computer keeps count of power on time in centiseconds in a 32 bit register. Just after 8 months elapses, the register overflows. Each computer goes into "safe" mode shutting down its generator resulting in a complete power failure, causing loss of control of the aircraft. The FAA Airworthiness Directive says to shut off the power before 8 months as the solution. SPARK/Ada would have prevented this.

NEW WORK

CubedOS

CubedOS is an operating system intended

for CubeSat flight control software. It will be used by Vermont Technical College in support of our Lunar IceCube work. However, the intent is for CubedOS to be general enough and modular enough for other groups to profitably employ the system. Since every mission uses different hardware and has different software needs, CubedOS is a really an application framework into which custom modules can be plugged to implement whatever mission functionality is required. CubedOS provides inter-module communication and other common services required by many missions. CubedOS thus serves both as a kind of operating system and as a library of useful tools.

CubedOS is written in SPARK with critical sections verified to be free of the possibility of runtime error. SPARK has also been used to provide some other correctness guarantees in certain cases. It is our intention that all CubedOS modules also be written in SPARK and proved free of runtime error (at least). However, CubedOS also allows modules, or parts of modules, to be written in full Ada or even C if appropriate. This allows CubedOS to take advantage of third party C libraries or to integrate with an existing C code base.

CubedOS can run directly on top of the hardware, with the assistance of a suitable Ada runtime system. It can also run as an ordinary process on top of a conventional operating system such as Linux, or on top of an embedded operating system such as VxWorks. This is made possible by the CubedOS low-level abstraction layer (LLAL). This layer plays a role in CubedOS similar to that played by the hardware abstraction layer used by many conventional operating systems. To port a CubedOS application to a new platform or underlying operating system, one should only need to provide a suitable LLAL.

The architecture of CubedOS is a collection of active and passive modules, where each active module contains one, and sometimes multiple, threads or tasks. Although CubedOS is written in SPARK/Ada there need not be a one-to-one correspondence between CubedOS modules and Ada packages. In fact, modules are routinely written as a collection of Ada packages in a package hierarchy, allowing complex modules to be implemented with the help of internal private child packages.

Critical to the plug-and-play nature of CubedOS, each active module is self-contained and does not make direct use of any code in any other active module (although passive modules serving as library components can be used). All inter-module communication is done through the CubedOS infrastructure with no direct sharing of data or executable content. In this respect CubedOS modules are similar to processes in a more conventional operating system. One consequence of this policy is that a library that several modules want to use must be either duplicated in each module or provided as an independent (passive) module of its own.

In the language of operating systems, CubedOS can be said to have a microkernel architecture where task management is provided by the Ada runtime system. Both low level facilities, such as device drivers, and high level facilities, such as communication protocol handlers or navigation algorithms, are all implemented as CubedOS modules. All modules are treated equally by CubedOS.

In addition to inter-module communication, CubedOS provides several general purpose facilities. In some cases only the interface to

the facility is described and different implementations are possible (even encouraged). Having a standard interface allows other components of CubedOS to be programmed against that interface without concern about the underlying implementation.

An asynchronous message passing system with mailboxes is used. This, together with the underlying Ada runtime system constitutes the "kernel" of CubedOS.
A runtime library of useful packages, all verified with SPARK:
A real time clock module.
A file system interface.
A radio communications interface.
Modules providing support for CCSDS protocols.
A CubedOS system also requires drivers for the various hardware components in the system. Although the specific drivers required will vary from mission to mission, CubedOS does provide a general driver model that allows components to communicate with drivers fairly generically. In a typical system there will be low level drivers for accessing hardware busses as well as higher level drivers for sending/receiving commands from components such as the radio, the power system, the camera, etc. The low level drivers constitute the CubedOS LLAL.

CubedOS provides several advantages over "home grown" frameworks.

The message passing architecture is highly concurrent and allows many overlapping activities to be programmed in a natural way. For example, our implementation of the CCSDS File Delivery Protocol (CFDP) takes advantage of this.
The architecture provides a lot of runtime flexibility; programs can adapt their communication patterns at runtime.
The architecture is consistent with the restrictions of Ada's Ravenscar profile.
CubedOS also brings several disadvantages over more customized solutions.

Because CubedOS messages are just octet sequences, there is runtime overhead associated with encoding and decoding them.
CubedOS sacrifices some static type safety; decoded messages must be validated at runtime with type errors being handled during the validation process. This is particularly worrisome in light of CubedOS's goal of providing robust assurances of correctness.
It is unclear at this time how analyzable CubedOS will be with the SPARK tools. We await access to SPARK 2014 tools that can process tasking constructs, which should be available in October, 2015.
CubedOS is an ongoing effort and should be considered experimental at this time. However, we hope to refine the architecture and implement enough non-trivial services to make CubedOS useful to other groups. Our long term goal is to distribute CubedOS to others working on CubeSat software or, for that matter, other similar embedded systems.

GEONS translation

We previously started a translation of NASA Goddard Space Flight Canter's GPS Enhanced Onboard Navigation System (GEONS) from its original C language to SPARK/Ada. We completed about 25% and found several errors using the SPARK tools, which we reported to Goddard. This project is on hold as we work on the Lunar IceCube software.

LUNAR ICECUBE

Collaborators

Lunar IceCube is a 6U (10cm x 20cm x 30cm, 14kg) satellite. It is manifested on the maiden flight of the NASA Space Launch System (SLS), Exploration Mission 1 (EM-1) in 2018.

Artists concept of the SLS Launch.

Morehead State University (Kentucky) is the principal investigator (PI) for Lunar IceCube under the Direction of Dr. Benjamin Malphrus. The science PI is Dr. Pamela Clark of Catholic University and NASA's Jet Propulsion Lab. The science instrument and navigation plan is from NASA Goddard Space Flight Center. The data/navigation radio is from the Jet Propulsion Lab.

Lunar IceCube Spacecraft description

Lunar IceCube, ion drive at bottom

Iodine ion drive

Lunar IceCube will use a Busek BIT-3 ion drive using solid iodine as the propellant.

Busek BIT-3 in operation.

This drive has a 3cm output grid, a 1.2 mN thrust and is mounted on gimbals for steering.

Computer

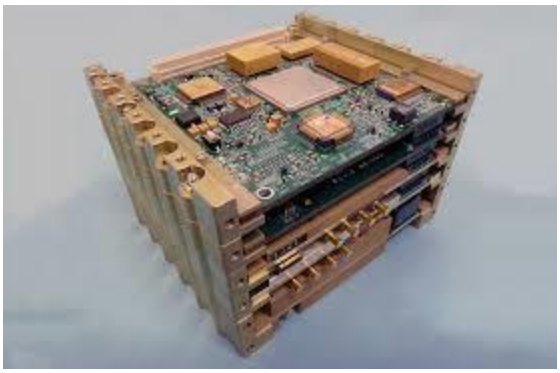
The main computer is a Space Micro Proton 400K. It uses a Freescale 2020 high performance processor, 1 Ghz, 32-bit (per core) dual core processor with 36-bit physical addressing. It is radiation hardened for operation beyond low Earth orbit.

Proton 400K CPU board

BIRCHES

The Broadband InfraRed Compact High Resolution Exploration Spectrometer from Goddard will be used to map water and other volatiles on the surface of the Moon.

Iris 2

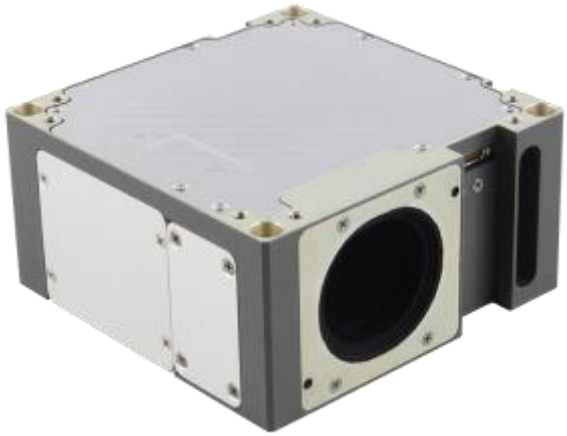


Iris 2 X-band radio.

Iris Version 2 is a CubeSat/SmallSat compatible transponder developed by the National Aeronautics and Space Administration's (NASA's) Jet Propulsion Laboratory (JPL) as a low volume and mass, lower power and cost, software/ firmware defined telecommunications subsystem for deep space. Iris V2's features include 0.5 U volume, 1.1 kg mass, 26 W DC power consumption when fully transponding at 5 W radio frequency output (8 W DC input for receive only), and interoperability with NASA's Deep Space Network (DSN supporting CCSDS standards and SLE data packet protocol. (the CCSDS file transfer protocol was the first SPARK/Ada software written for Lunar IceCube by our students) at X-Band frequencies (7.2 GHz uplink, 8.4 GHz downlink) for command, telemetry, and navigation.
The radio supports a wide range of data rates needed for wide range of distances, and has full duplex capabilities appropriate for Doppler and ranging. Iris 2 utilizes a Gaisler LEON3-FT softcore (on Virtex 6). The system produces 37 dBm transmit power with-130 dBm receive sensitivity.

ADACS

XACT star tracker/momentum wheels



We are using a Blue Canyon Technologies XB-1 system, consisting of two XACT Micro Star Tracker and Micro Reaction Wheel assemblies. XACT features 3-axis Stellar Attitude Determination in a micro-package. Multiple reference frames: Inertial, LVLH, Earth-Fixed, and Solar. Precise 3-axis control is provided by low jitter reaction wheels, torque rods and integrated control algorithms.

PV panels

Power will be supplied with a 120W (in two panels) MMA E-HaWK™ Deployable Solar Panel Array, both of which are aimed with Honeybee Robotics solar array drives.

MISSION

Transfer to Lunar orbit

Launch using EM-1: Lunar IceCube will be injected into a direct transfer as a payload onboard EM-1. Our mission profile uses the Interim Cyrogenic Propulsion Stage (ICPS) disposal state that occurs at approximately 1-hr after the EM-1 injection as our initial condition.
This state, if unchanged, would result in a lunar gravity assist at an approximate lunar

altitude of 1300-km and enter into a heliocentric drift-away orbit. We will use the described low-thrust propulsion system to modify the trajectory derived from the supplied state in order to change the first lunar flyby to different B-Plane components with a radial distance of 9239 km, permitting a post lunar flyby design that incorporates dynamical systems (manifolds) and minimizing the total required delta-v.

Collecting data

Data will be collected from the BIRCHES instrument and stored, as during operation, the BIRCHES end of the spacecraft must point to the Lunar zenith.

Data download

After a data collection pass, the spacecraft will be rotated by the ADACS so that the patch antennas will point toward the Earth for communication with the DSN.

Lunar IceCube software

As can be seen from the great complexity of the spacecraft systems and operations, the software for Lunar IceCube will be very substantially more complex than our current CubeSat software. We have proven the utility of the high integrity software technology of SPARK/Ada in that of the 12 university CubeSats we were launched with, we are the only one operational. We are looking at several software design tools to help with the overall design, such as UML and Simulink based. Our current Cubesat software was created in a rather ad hoc fashion, but the complexity of Lunar IceCube will require a much more structured approach.

End - Moon or Mars?

At the end of the data collection period of about six months, the spacecraft cannot be left in Lunar orbit. With no atmosphere, it would remain there indefinitely. We have permission to dispose of it by using the ion drive to deorbit and crash into the Moon. Another option, if volume considerations allow the full initial design quantity of iodine (a 350cc tank, holding 1.75kg of iodine), we would have enough delta-v, 2,800 ms^{-1}, when only about 850 ms^{-1} is needed for the Lunar mission) capability to leave Lunar orbit and head to Mars. The final experiment would be to see how long we could remain in radio contact with Lunar IceCube.

References

[1] *Building High Integrity Applications with SPARK*
by John W. McCormick & Peter C. Chapin
Cambridge University Press
ISBN-10: 1107656842
ISBN-13: 978-1107656840

[2] ALT-ERGO: http://alt-ergo.lri.fr

[3] CVC4: http://cvc4.cs.nyu.edu/web/

[4] Microsoft Z3:
https://github.com/Z3Prover/z3

Further SPARK information:

http://libre.adacore.com/tools/spark-gpl-edition/

http://www.spark-2014.org